

LINUX EXPERT

FAQ

Q Backups using tar take ages, so I usually put them in the background by adding '&' to the end of the command. Is there an easy way to check they have finished?

A People tend to monitor a running tar process with a piped command such as `ps -ax | grep tar`. But you have to keep running this command to see what's going on. Luckily, there's an automated alternative.

The `watch` utility, supplied with most Linux distributions, will clear the screen and re-run a command and show its output on a regular basis. So use:

```
# watch 'ps -ax | grep tar'
```

By default, `watch` will run the command that follows it in single quotes every two seconds. You can use the `-n` switch on the command line to alter the default interval between refreshes.

Another problem is monitoring growing log files. There's a flag to the `tail` command that will display the end of a file repeatedly. Try the `-f` switch, useful for monitoring steadily growing log files. The default output is the last 10 lines, but you can change this with the `-n` switch too:

```
# tail -f -n 25 /var/log/messages
```

Jon Thompson
UNIX/Linux management
and security consultant



linux@computershopper.co.uk

Make Apache secure

Do you find yourself agonising over the security of your web server? Jon Thompson explains the security measures you can apply in Apache to keep those pesky hackers at bay

Do you truly believe that your Apache installation and the scripts it runs are secure? You may have installed a solid firewall and intrusion-detection system, but there are still ways to make your web server misbehave. In this month's *Linux Expert*, we show you how to thwart hackers with the minimum of effort and without rewriting all your scripts.

Some attacks masquerade as perfectly reasonable browser requests, but they can also allow a skilled hacker in to damage your systems, steal data and even launch attacks against others. Your server and the scripts it runs need to be bulletproof to face the internet. But heavy-duty programmers who know how to write securely are expensive. One alternative is to use some of the tempting pre-written scripts available online that allow your site to do everything from user registration to credit card processing. But with so much on offer, not even a Linux expert can be sure that what he runs is completely safe. Luckily, there's an ingenious solution that will soon have you sleeping soundly again.

COUNT THE WAYS

Before launching an attack against a web server, a skilled hacker will spend time working out what you're running and what defences are in place. Unlike other services, web servers allow a wide range of interaction. The hacker might look at the headers returned from the server to discover the version of Apache you're running. If it's out of date, he might have an effective exploit. He can also attack your home-grown scripts, which are likely to be more vulnerable than Apache's code.

He will visit every page on your site and note down what appears in his browser's address bar. From this, he can work out the structure of the site and its scripts. He might then send malformed page requests that return specific errors to test just how much checking the scripts perform. He'll keep going, looking for any scrap of information that your site will give him by mistake. The biggest threat comes from pages into which the user can input information. So the next line of defence is to ensure that web pages use the correct method of collecting data.

There are two main methods of obtaining web input. With the GET method, the browser itself packages data into the next URL the user requests. Data sent this way appears after a '?' in the URL. It's easy to edit this string and resubmit it to see what happens. For example, `www.site.com/article.php?id=1` could load the first article in a database, but a hacker could type `www.site.com/article.php?id=unexpected`. How the site behaves will depend on how the script is written.

The more secure POST method, on the other hand, splits out the data returned by the browser and sends it as a separate transmission called the body, which does not appear in the URL. This makes the data fairly safe from the casual mischief-maker, because it demands that any successful hacker intercept and change this hidden body. It also means that the focus for your

security efforts now rests upon what happens when the request finally arrives at the server. Because of this, the range of checks and balances even simple scripts must incorporate has to cover lots of potential threats.

Imagine a website with an online catalogue, on the main page of which users can enter a part number to check a product's availability. A PHP script in `/var/www/cgi-bin` takes that part number and calls a MySQL database, which runs a search for the part and finally returns its details and the associated stock level. If the part number is wrong, the database will return an error, which the script should handle to produce the appropriate output. This might seem foolproof, but there are problems.

What happens, for instance, if the user doesn't enter a part number but types in a string of other characters? In this example, the script still passes the request to the database, even though it's not a part number. If the user has added certain characters at the start of a string, he can make what follows these characters be interpreted as SQL statements by the database.

This will return data far more interesting than a simple stock level. Hackers know this form of attack as SQL injection. Depending on what your database stores, SQL injections can return anything from email addresses to credit card details. See the box Lethal injection opposite for more on this.

POSITIVELY NEGATIVE

Capturing all instances of all types of attack is a mammoth task for our simple script, but there are two basic security models we can use. They're not just limited to web server scripts, either; firewalls use them too. First, there's the popular negative security model. In this model, you identify every dangerous pattern of use and configure the script to reject them all. This is as simple as programming a long list of 'if... then' statements, but it also requires extensive coding to cover all types of attack.

The second security model is called the positive security model, also known as the white list model. The developer seeks to identify only those actions that are acceptable and rejects all others. This is inherently safer than the negative security model, but needs more thought for complex scripts. One might accept only requests that have parameters in the correct range, for instance, with acceptable characters, content length and type.

Though this approach is the most effective, our hypothetical script would need careful planning and testing so as not to trap and reject legitimate user requests. To help website programmers avoid spending all their time writing paranoid code, there have been several attempts to integrate the positive security model into an existing intrusion detection system (IDS). Web servers can produce highly detailed logs, so we could use an IDS to search these for patterns of known activity and determine whether an attack is under way.



However, your IDS might only be telling you that you've already handed your user database to a stranger.

A better option would be a real-time system that inspects incoming requests and makes a decision to reject or allow them before they hit the web server. Anything it doesn't like, it rejects. It could check for different classes of suspicious web activity so that developers can get on with the business of building content that stretches the power of today's web servers and gives users the best experience possible. Luckily, such a system does exist, and it's free and effective.

ENTER THE MODFATHER

ModSecurity is a highly programmable positive security module for Apache that intercepts, scrutinises and rejects any requests it deems unacceptable. It does the same for the responses sent from the server, so even if an innocent-looking request turns out to be malicious, there's little chance of the server giving out forbidden information. ModSecurity can also produce a full audit trail of its actions and the attempted security violations it has caught.

To install ModSecurity, you'll need the gcc compiler that came with your distribution and the Apache development package. This has the name Apache2-devel-nn, where nn is the version number. If you have a graphical installation tool such as YaST, installation of both packages is easy. If not, or if you want to download the latest versions of the RPMs, you can install them with the following command:

```
# rpm -i <package name>#
```

The Apache development package is important because it gives access to a utility called apxs2, which we need so we can build ModSecurity's mod_security module and add it to the Apache installation. We're not going to cover installing ModSecurity on Apache 1.3 as this version has largely been superseded by version 2.

ModSecurity doesn't yet come as standard with all Linux distributions, so you'll probably have to download the source tarball from www.modsecurity.org/download. At the time of writing, the current version is 1.9.2. Unpack the downloaded tarball as follows:

```
# tar zxvf <tarball name>#
```

Navigate to the Apache2 subdirectory inside the unpacked tarball:

```
# cd modsecurity-apache-1.9.2/
  apache2#
```

You can now build the module as follows:

```
# apxs2 -cia mod_security.c#
```

After a couple of pages, apxs2 will finish.

You should now access the file /etc/sysconfig/apache2 to ensure the installation process has added the module to its list. The list APACHE_MODULES should include the word security. If it does not, which is likely, open an editor such as vi and add it by hand. Now you can start to configure the server.

CONFIGURATION

Like other Apache configuration directives, ModSecurity commands live in the httpd.conf

file, which is stored in the /etc/apache2 directory. The commands might be called into httpd.conf from another file if httpd.conf contains any Include directives.

ModSecurity's documentation hints that, once installed, simply restarting the server might cause a core dump to occur. In order to prevent this, use the /etc/init.d/apache control script to stop and then start the server, or use the commands:

```
# apache2ctl stop#
# apache2ctl start#
```

Here's an initial list of common ModSecurity configuration directives. We'll run through each of these in turn, describing what they are and what they do:

```
<IfModule mod_security.c>#
SecFilterEngine On#
SecFilterScanPOST On#
SecFilterCheckURLEncoding On#
SecFilterCheckUnicodeEncoding On#
SecFilterForceByteRange 32 126#
SecAuditEngine On#
SecAuditLog /var/logs/audit_log#
SecFilterDebugLevel 4#
SecFilterDebugLog /var/www/log/
  modsecurity_debug_log#
SecFilterScanOutput On#
SecAuditLogRelevantStatus ^5#
SecFilterDefaultAction deny, log,
  status:403#
</IfModule>#
```

We want the server to obey ModSecurity directives only when installed and active, so we place them within the block statements <IfModule mod_security.c> and </IfModule>.

• SecFilterEngine On

The first directive within the block enables ModSecurity protection. The parameters here are On, Off and DynamicOnly. The last one ensures that ModSecurity responds only to the output of scripts rather than laboriously checking static HTML content too. Once active, we need to set up a few global rules for ModSecurity to abide by.

• SecFilterScanPOST On

As mentioned earlier, it's a good idea to use the POST method to transmit data to the server, so we need to examine all data sent via this method. The SecFilterScanPOST scans these bodies explicitly, as well as other requests.

• SecFilterCheckURLEncoding On

• SecFilterCheckUnicodeEncoding On

The system should examine any special characters that the browser converts to Unicode before transmitting them to the server. In the URL, Unicode characters have the format %AB where AB is a pair of hexadecimal digits. Careful manipulation of these characters in the URL is one way hackers can introduce sequences that do things such as trying to break out of the directory structure that contains your website. Sequences that decode as ../ in the URL, for instance, might be able to navigate out of /var/www and into a lower level of the file system.

Lethal injection How hackers extract data from a MySQL database

One technique beloved of website hackers is the SQL injection. This is where an attacker, instead of entering a sensible value into an input box, sends a sequence of characters that changes the meaning of a query constructed and passed by PHP to MySQL.

To explain, consider the following line from a PHP script:

```
$query = "SELECT * FROM users WHERE
  username='$username' AND password =
  '$password'";
```

When we enter a username of 'jonthompson' and a password of 'nitwood', the string stored in \$query becomes:

```
SELECT * FROM users WHERE
  username='jonthompson' AND
  password='nitwood'
```

This asks the server to return all rows that match the values for the entered username and password. The script can then check the number of rows returned. If it's zero, the script knows the user entered an invalid username or password. If it's exactly one, he must have entered the correct credentials.

The problem is that a sneaky attacker can enter characters that cause a naïve PHP script to create a SQL statement that removes the password check, simply by entering the following as the username:

```
Jonthompson' --
```

The single quote character after the username terminates the current statement in SQL. We've also added a double minus, which tells SQL to comment out the rest of the line, which contains the password test. Once assembled by PHP, the new SQL statement in \$query looks like this:

```
SELECT * FROM users WHERE
  username='Jonthompson' -- 'AND
  password='nitwood'
```

This is the same as saying:

```
SELECT * FROM users WHERE
  username='Jonthompson'
```

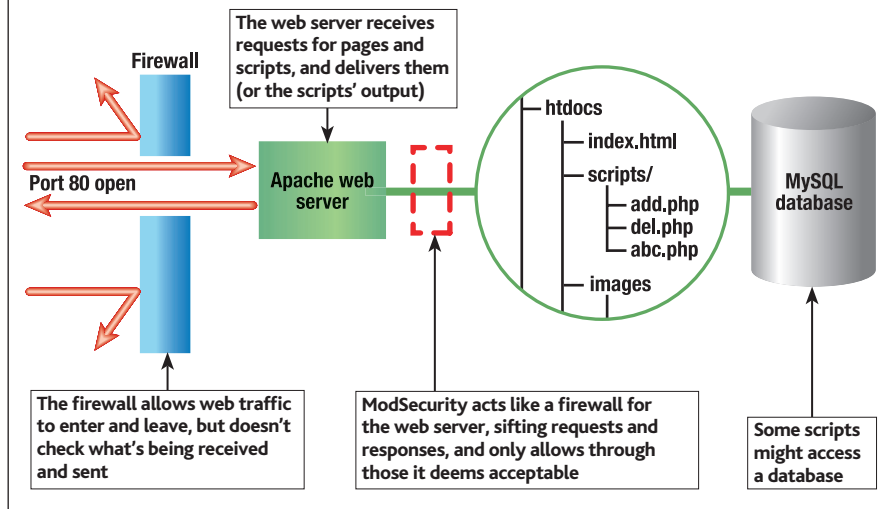
So we've changed the meaning of \$query and, in doing so, removed the need for a valid password. An attacker could insert more SQL statements before the final '--' perhaps to add a new user or possibly even delete data.

To guard against SQL injection attacks, you can use the PHP strip_tags function:

```
$username = strip_tags($_
  POST['username']);
$password = strip_tags($_
  POST['password']);
```

This removes all instances of statement terminators, comments and characters that can change the meaning of the subsequent query, and should show an attacker that you're not interested in his attentions.

Positive security in action How ModSecurity works



Some Unicode character sequences simply confuse the server and cause it to crash or produce information the hacker can subsequently use to tailor his attacks. To prevent variations on this theme, there are a couple of ModSecurity directives to ensure that URLs sent to the server aren't malicious.

When combined, `SecFilterCheckURLEncoding` and `SecFilterCheckUnicodeEncoding` check that the Unicode used is sensible and that the characters that are used in the URL do not form any malicious sequences.

• `SecFilterForceByteRange 32 126`

There's a further directive we can use to ensure no dodgy characters appear in the URL or POST data. In our example database lookup form, we want to accept a plain text part number. To define which characters are acceptable, we can specify the range of characters we want ModSecurity to pass using this directive. Character 32 is the first printable ASCII character (a space) and character 126 is the last (a tilde or '~').

With our configuration so far, whenever the `SecFilterEngine` directive is set to `On` or `DynamicOnly`, our server will accept only well-behaved URLs containing printable characters, and no funny business that a hacker might have planted manually.

• `SecAuditEngine On`

• `SecAuditLog /var/logs/audit_log`

It's time to add some directives to log debugging and audit information. If your server is the subject of an attempted attack, a full audit trail of the hacker's actions is a great boon. Turn on auditing and tell ModSecurity where to store the audit trail. It's worth noting what you can expect to see in the audit log:

```
=====
Request: 192.168.0.2 - - [[14/Feb/2006:11:34:11
+0000]] "GET /cgi-bin/printenv?p1=666 \
HTTP/1.0" 406 822
Handler: cgi-script
-----
GET /cgi-bin/printenv?p1=666 HTTP/1.0
Host: 192.168.0.1:80
User-Agent: mod_security
Connection: Close
mod_security-message: Access denied with code
```

```
406. Pattern match "666" at \
ARGS_SELECTIVE
mod_security-action: 406
HTTP/1.0 406 Not Acceptable
=====
```

• `SecFilterDebugLevel 4`

• `SecFilterDebugLog /var/www/log/modsecurity_debug_log`

Set the level of debugging information ModSecurity produces while you test it, and the file to which you log.

Make sure the directory `/var/www/logs` exists. These directives should be self-explanatory. The first tells ModSecurity the level of debugging information you want to collect; the second defines where to log it. The format of the debug log is the same as other syslog files. The levels of debugging information are as follows:

0	No debug logging (useful for tested production systems)
1	Log only significant debugging events
2	Give informational messages about general events
3	Produce detailed informational messages
4	Full debugging information

• `SecFilterScanOutput On`

As mentioned earlier, ModSecurity can also filter the output from the server, in which case all output is subject to its rules. The `SecFilterScanOutput` directive enables you to define rules about what to do when the server gives certain responses, further frustrating your would-be attacker.

• `SecAuditLogRelevantStatus ^5`

With `SecFilterScanOutput` enabled, you can log response codes from the server, for example. This example logs all codes starting with 5 (500 is an internal server error and may be generated by a malicious attack).

• `SecFilterDefaultAction deny, log, status:403`

The last directive defines the default action to take in cases where ModSecurity intercepts something it believes to be malicious but has no explicit rule for handling it. In this case, the visitor gets a Forbidden error message and the system logs his access attempt.

MAKING THE RULES

Once ModSecurity is configured, we can design some rules for it to follow when it receives a request. ModSecurity has a flexible rules engine, which filters and acts upon requests before they ever reach the server. When a rule finds a match, it can take any of a number of actions. We'll explain the basics of this process, then give you some useful examples to show the types of things you can do to frustrate a would-be hacker.

The simplest way to filter incoming requests is to use a set of `SecFilter` directives, which appear like this:

```
SecFilter <KEYWORD>
```

This directive instructs ModSecurity to look for `<KEYWORD>` in the incoming request and obey our previously defined `SecFilterDefaultAction` if found. In this case, it will deny access to the requested resource, log the attempt and return a status of 403 (forbidden resource) to the user's web browser. The expression `<KEYWORD>` is a regular expression, which means you can define it precisely. A good introduction to the use of regular expressions is available at http://en.wikipedia.org/wiki/regular_expression.

`SecFilter` is limited in its scope and can introduce false positives that might put off potential users, so there's also the `SecFilterSelective` directive, which enables you to apply rules to specific parts of a request or an incoming POST body with more accuracy. The format for a selective rule is as follows:

```
SecFilterSelective <VARIABLE>
<KEYWORD> <ACTIONS>
```

`<VARIABLE>` is one of a large number of built-in variables that you can use to search selectively for the regular expression `<KEYWORD>` in the incoming request. We don't have space to replicate the entire list of variables here, but you can find it at <http://tinyurl.com/8by3x>. The list of variables covers everything from the incoming IP address to the time of the request, which should give you ample scope for controlling access to your site. Usefully, it dynamically includes the names of any variables used in the forms on your web pages, which we'll use in the examples that follow.

`<KEYWORD>` is a regular expression as before. In this case, `^$` matches an empty string. `<ACTIONS>` is one of a comprehensive list of actions. You can combine actions by separating them with commas, as we've done with `SecFilterDefaultAction` above. A list of these is available at <http://tinyurl.com/8gckv>, but three are particularly important. The so-called 'primary' actions are `deny`, `pass` and `redirect`. `Deny` and `pass` reject or allow a browser's request for a resource, but `redirect` is more flexible. It enables you to substitute a resource, such as a web page or script, for the requested one. This could give a message indicating that you suspect the surfer is a hacker and you have logged their actions and IP address, for example. The format is `redirect: <url>`, where `<url>` is the path to the resource that you want the user to see instead of the one requested.

You can also use the `|` pipe symbol to specify more variables in a rule, making the mechanism very flexible. For example:

```
SecFilterSelective HTTP_User-
```



```
Agent|HTTP_Connection|HTTP_Accept ^$
log, pass
```

When someone tries to connect a Telnet client to port 80 and requests information about the server, which is legal but highly suspicious, they won't normally send information that goes into the three variables named in the above rule. In this case, the rule will let the request pass, but will log it too. Instead of logging to the debug log, you can use the auditlog action to log the request to the audit log, if you have one set up.

USEFUL EXAMPLES

Suppose you have a web page (let's call it add_user.php) that contains a form. This allows a new user to register his details and become part of your website. This is a simple task, yet it is wide open to abuse from a bewildering number of angles. We want to ensure that the server accepts only data sent using the POST method, that these requests include only the user's first and second name, along with their email address, and that the maximum length of any field is 64 characters. This length will stop buffer overflow attacks of the type we explored in *Linux Expert*, *Shopper* March 2006. This many characters should be enough to accommodate even the longest of names.

We'll also insist that the last name be filled in with a value, and that the email address must contain only characters normally found in an email address. With a perusal of the regular expression rules, we'll create a set of ModSecurity rules to cope with our needs that apply only to the script we're protecting:

```
<Location /add_user.php>
SecFilterSelective REQUEST_METHOD
  !^POST$
SecFilterSelective ARGS_NAMES !^(first
  name|lastname|email)$
SecFilterSelective ARG_firstname !^[[:
  alnum:][:space:]]{1,64}$
SecFilterSelective ARG_lastname !^[[:
  alnum:][:space:]]{1,64}$
```

```
SecFilterSelective ARG_email
  !(^$|^[[:alnum:]].@]{1,64}$)
</Location>
```

One of the benefits of ModSecurity is that the ARG variables in this example use the names of the variables as they appear in the original add_user.php page. ARGS_NAMES contains a list of them in the order they appear on the page. This set of directives should ensure that only those credentials that meet our requirements ever reach the server. Because there's no explicit action at the end of these directives, ModSecurity passes those that don't conform to the regular expressions to the SecFilterDefaultAction directive for action.

Some legitimate website security testing tools can be used maliciously by the wrong hands. To protect against such tools, it's a good idea to log their use. Unknown to many inexperienced hackers, tools such as Whisker announce their identity to the server by default. To capture the use of many of the most popular tools quietly, you could use the following rule:

```
SecFilterSelective HTTP_User-Agent
  (libwhisker|whisker|paros|wget|libww
  w|perl|curl) log, pass
```

Or you could simply 'deny' rather than pass the request, if one of these tools comes calling.

There are times when even in the most secure of environments, the administrator's credentials fall into the wrong hands. When running a web-based message board, for instance, this might mean anyone can edit or even remove posts and users. To prevent others logging in despite having the administrator's username and password, try adding the following to your web-based login page within a <Location> block:

```
SecFilterSelective ARG_username
  ^administrator$ chain
SecFilterSelective REMOTE_ADDR
  ""^192.168.0.1$" allow
```

This pair of rules first tests to make sure the username variable passed from the browser contains the 'administrator' username, assuming this is the name of the admin account. If it isn't, simply substitute the real one and the real variable name. The action it takes is 'chain', which causes ModSecurity to execute the second directive only if the username is administrator.

The second directive allows the request to proceed if the source address is that contained within REMOTE_ADDR. Make this an internal IP, such as the address of the server itself, and then no-one, either on the internal network or the internet in general, can log in if they're not physically sitting at the console.

There are also some fun things you can do with ModSecurity to distract hackers and lead them on a merry dance. Responses usually start with the name and version of the server. Using ModSecurity, you can fake this string to mask the fact that you're running Apache. Even if someone connects a real web browser instead of Telnet and uses tcpdump to examine the information received back, they'll receive nonsense. This will confuse automated attack tools, too, so they fire off a volley of useless hacks that are effective only against other platforms. The following will make your Apache server appear as a Microsoft IIS 5.0 server:

```
SecServerSignature "Microsoft-IIS/
5.0"
```

ModSecurity is a module that enables the average PHP programmer to concentrate on adding functionality to a website and not worry about whether he's coded for all possible attacks. It also ensures that, with a bit of thought, would-be attackers quickly get the message and try elsewhere.

In a world where hacking shows no sign of going out of fashion, it's essential that you make your website and the underlying server as unfriendly as possible to all but legitimate users, and ModSecurity will help you achieve this.

Apache in jail Using chroot to lock a server into a directory

It's almost a given that someone, somewhere, can break into the most secure software installations in the world. However, there are ways to lessen the impact of a successful hack. One such method is a security technique that's known as chrooting, which seeks to make an application and its files run in a different place from its installation directory.

The chroot command, from which the name of the technique comes, enables you to put a complete server into a directory from which it cannot break free. Therefore, even if a hacker compromises it, it cannot be used as a springboard to further exploit the system. Such chroot jails are very secure if done properly, but you need to copy the entire application and its files into a new file structure and start it with the chroot command.

Unfortunately, there are as many tutorials about breaking out of chroot jails as there are tutorials about setting one up. Luckily, ModSecurity can come to the aid of the

security-conscious Apache server administrator.

The annoying thing about traditional chrooting techniques is that all the libraries and other files the application needs must also go into jail, because the application cannot access anything outside its new directory tree.

In some circumstances, however, where the application accesses its files just once during initialisation, you can place the running process into the jail dynamically, after it accesses whatever it needs. This is dodgy to do manually, but there is a chrooting facility built into the mod_security module that makes the whole process feel like it's much less complicated to execute. First, you need to set up a chroot jail by using the following commands:

```
mkdir -p /chroot/apache/usr/local
cd /usr/local
mv apache /chroot/apache/usr/local
ln -s /chroot/apache/usr/local/
apache
```

This assumes that Apache is installed in /usr/local on your Linux distribution. You now need only to add one line to the configuration file to lock the door to the Apache chroot jail when it starts, after it has read its configuration files:

```
SecChrootDir /chroot/apache
```

Apart from this simplicity, ModSecurity chrooting brings another advantage. Unlike a traditional chroot jail, the ModSecurity method requires no additional files to be in jail. All shared libraries are already loaded, all modules are initialised, and the log and audit files are open.

Apache control changes slightly when you use ModSecurity to handle your chroot jail. To start and stop the server, you need to use the apachectl2 command found in the jail, such as:

```
/usr/local/apache/bin/apache2ctl
start
```